ValueExpert: Exploring Value Patterns in GPU-Accelerated Applications

Keren Zhou*
keren.zhou@rice.edu
Rice University
Houston, Texes, USA

Yueming Hao*
yhao24@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

John Mellor-Crummey
johnmc@rice.edu
Rice University
Houston, Texes, USA

Xiaozhu Meng
xm13@rice.edu
Rice University
Houston, Texes, USA

Xu Liu
xliu88@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

ABSTRACT

General-purpose GPUs have become common in modern computing systems to accelerate applications in many domains, including machine learning, high-performance computing, and autonomous driving. However, inefficiencies abound in GPU-accelerated applications, which prevent them from obtaining bare-metal performance. Performance tools play an important role in understanding performance inefficiencies in complex code bases. Many GPU performance tools pinpoint time-consuming code and provide high-level performance insights but overlook one important performance issue—value-related inefficiencies, which exist in many GPU code bases. In this paper, we present ValueExpert, a novel tool to pinpoint value-related inefficiencies in GPU applications.

ValueExpert monitors application execution to capture values produced and used by each load and store operation in GPU kernels, recognizes multiple value patterns, and provides intuitive optimization guidance. We address systemic challenges in collecting, maintaining, and analyzing voluminous performance data from many GPU threads to make ValueExpert applicable to complex applications. We evaluate ValueExpert on a wide range of well-tuned benchmarks and applications, including PyTorch, Darknet, LAMMPS, Castro, and many others. ValueExpert is able to identify previously unknown performance issues and provide suggestions for nontrivial performance improvements with typically less than five lines of code changes. We verify our optimizations with application developers and upstream fixes to their repositories.

CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures**; • **Computing methodologies** → **Parallel programming languages**; **Shared memory algorithms**.

*Both authors contributed equally to this research.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507708

KEYWORDS

GPUs, GPU profilers, Profiling Tools, Value Analysis, Value Patterns

ACM Reference Format:
Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. 2022. ValueExpert: Exploring Value Patterns in GPU-Accelerated Applications. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3503222.3507708

1 INTRODUCTION

GPU architectures have become mainstream in modern and emerging computing systems to accelerate applications in different domains. For high performance computing (HPC), the GPU-accelerated Summit supercomputer is ranked #2 among the 500 most powerful supercomputers in the world in June 2021 [1]. Moreover, all of the emerging exascale supercomputers developed by the Department of Energy (DOE) will be accelerated with GPUs [12]. For cloud computing, Amazon Web Service provides GPU-based compute instances [6]. For embedded computing in autonomous systems, NVIDIA Jetson [33] integrates CPUs and GPUs.

GPUs typically employ thousands of cores and high-bandwidth memory to enjoy massive parallelism and high performance. To facilitate programming on GPUs, emerging programming languages (e.g., CUDA [32], OpenMP [38], Kokkos [15], and RAJA [9]), compilers (e.g., NVCC [37] and LLVM [27]), and frameworks (e.g., Tensorflow [4], PyTorch [39], and AMReX [57]) provide various interfaces to offload computations to GPUs. However, due to the complexity of GPU architectures, it remains challenging to write efficient programs to harness the compute power of GPUs. Performance inefficiencies can hide deep in GPU code, preventing applications from obtaining bare-metal performance.

1.1 A Motivating Example

Darknet [44] is a popular deep learning framework written in CUDA and C. Darknet's cuBLAS [29] backend implements convolution using the lowering method [20]. We study Darknet using the YOLOv4 [11] neural network, identify two representative inefficiencies, and propose optimizations that address them without accuracy loss.

Inefficiency I: redundant GPU instructions. In the forward phase of each convolution layer, function `fill_ongpu` (Line 2) sets array

```
1forward_convolutional_layer_gpu(...) {
2  fill_ongpu(l.outputs * l.batch, 0, l.output_gpu, 1);
3  for (j = 0; j< l.groups; ++j) {
4    gemm_ongpu(..., 1, l.output_gpu);
5    ...
6  }
7}
```

Listing 1: A redundant invocation of GPU kernel `fill_kernel`. Changing the argument `1` to `0` of `gemm_ongpu` can remove read operations on array `l.output_gpu`.

```
1convolutional_layer make_convolutional_layer(...) {
2  l.output_gpu = cuda_make_array(l.output, total_batch * out_h *
       out_w * n);
3  l.x_gpu = cuda_make_array(l.output, total_batch * out_h * out_w
       * n);
4}
```

Listing 2: Unnecessary CPU-GPU communication in Darknet. Darknet copies CPU array `l.output` (initialized to zeros) to GPU arrays `l.output_gpu` and `l.x_gpu`.

`l.output_gpu` to zeros as shown in Listing 1. Function `gemm_ongpu` then updates `l.output_gpu` by reading its values and accumulating them across all iterations. When there is only one iteration, we can remove `fill_ongpu` and associated read operations in `gemm_ongpu`, which reduces each convolution layer's loads and stores executed on the GPU by 4.1% and 10.6%, respectively.

Inefficiency II: unnecessary CPU-GPU data transfer. In function `make_convolutional_layer`, as shown in Listing 2, Darknet initializes `l.output`, an array on CPUs, to zeros via function `xcalloc`. Darknet then copies `l.output` to `l.output_gpu` (Line 2), an array on the GPU. This copy on zeros wastes memory bandwidth. The same problem exists in other arrays, e.g., `l.x_gpu` (Line 3). It is better to use `cudaMemset` to directly initialize these arrays on the GPU side instead of copying zeros from the CPU, which saves 84.2% CPU-GPU memory traffic.

1.2 Existing Solutions

Profilers play an important role in bridging the gap between software and hardware by identifying performance inefficiencies. There exist many GPU profilers, including vendor-provided tools such as Nsight Compute [35], Nsight Systems [36], nvprof [34], VTune [45], and AMD ROC-profiler [2], as well as open source tools such as HPCToolkit [59], TAU [28], and Score-P [25]. These profilers pinpoint hot GPU code via measuring elapsed time or hardware events.

However, these profilers cannot easily identify the inefficiencies we found in Darknet or provide intuitive guidance. Hotspots often show up as symptoms of performance inefficiencies; analyzing their root causes typically requires significant manual effort. Without knowledge of root causes, one cannot easily optimize Inefficiency I and II even if one knows they are inefficient because of the following limitations of existing profilers:

- *Lacking a microscopic view:* Existing profilers typically collect a limited set of performance metrics using hardware counters. Without a microscopic view of the behavior of individual instructions, one cannot easily identify and optimize many performance inefficiencies, such as Inefficiency I.
- *Lacking a holistic view:* Existing profilers mostly target individual GPU API invocations and provide little insight into interactions across multiple GPU APIs. Their myopic views miss many optimization opportunities, e.g., Inefficiency II.

1.3 Our Approach

In this paper, we present ValueExpert, a novel value profiling and analysis tool to identify value-related inefficiencies, such as the inefficiencies shown in Section 1.1. As a unique feature, ValueExpert provides microscopic value pattern analysis and global value flow analysis to obtain deep insights for code optimization.

- *Microscopic value pattern analysis:* ValueExpert leverages binary instrumentation to monitor memory load and store operations in GPU kernels and capture the values used or produced by these operations. Furthermore, ValueExpert associates values with data objects (e.g., arrays or tensors) to identify various value patterns at both coarse- and fine-grained levels, which guides actionable optimization.
- *Global value flow analysis:* ValueExpert tracks value flows for data objects in a global view: across CPUs and GPUs, as well as across GPU API invocations. The value flows include object allocations, initializations, transfers, uses, and updates. ValueExpert constructs a value flow graph to broaden the scope of inefficiency analysis beyond individual GPU API invocations.

The implementation of ValueExpert addresses challenges in collecting, maintaining, analyzing, and presenting a large volume of performance data. First, ValueExpert needs to handle the high parallelism and limited memory on GPU to efficiently store fine-grained performance data with minimum interference to GPU kernels. Second, ValueExpert needs to minimize data transfer between GPU to CPU when the profiling data fills the allocated memory on GPU. Third, ValueExpert needs to employ a novel visualization technique to present and analyze the massive and detailed performance data.

We address these challenges and develop ValueExpert for mainstream systems accelerated with NVIDIA GPUs, including commodity Linux clusters with and multiple compute nodes with multiple GPUs per node. ValueExpert monitors fully optimized executables without source code modification or recompilation required. We apply ValueExpert to optimize deep learning frameworks (e.g., PyTorch [39], Darknet [44]), and important HPC applications (e.g., LAMMPS [42], NAMD [41]), and a well-known GPU benchmark suite—Rodinia [13]. ValueExpert successfully identifies prior-unknown performance bugs in these applications with moderate overhead on two NVIDIA GPU platforms: RTX 2080 Ti and A100. Guided by ValueExpert, we are able to obtain 1.58× and 1.39× geometric mean speedups for applications running on RTX 2080 Ti and A100 accordingly, with typically less than five lines of code changes. To verify the correctness and significance of ValueExpert's findings, we either confirm with the application developers or upstream our optimization patches to application repositories.

Contribution Summary.

(1) We categorize eight value patterns in GPU-accelerated applications and discuss optimization opportunities by exploiting each pattern.
(2) We design and implement ValueExpert—a value profiling and analysis tool that recognizes value patterns and construct value flows to pinpoint value-related inefficiencies.
(3) We describe novel parallel analysis algorithms on GPUs that accelerate ValueExpert.
(4) We enable ValueExpert to provide rich information, including full call paths for each GPU API and value flow graphs, to help users identify optimization opportunities.
(5) We apply ValueExpert to analyze several production ML and HPC applications, achieving nontrivial speedups. We upstream our optimizations to benefit the community.

Limitations. First, ValueExpert is a dynamic analysis tool; it requires inputs that trigger execution behaviors of interest. second, ValueExpert currently works on NVIDIA GPUs only but its methodology is generally applicable to GPUs from other vendors if necessary binary instrumentation engines are available. Third, ValueExpert is a profiler. It pinpoints and analyzes inefficiences, but does not automatically fix them. Additionally, although ValueExpert does not have any false positives in identifying value patterns, programmers are responsible to apply profitable optimization.

2 RELATED WORK

Classical GPU performance tools [2, 18, 25, 28, 34, 35, 45, 59, 60] profile or trace GPU activities. Unlike ValueExpert, none of these tools analyze value-related inefficiencies. GPU simulators [8, 22, 43] monitor execution details, but incur prohibitively high overhead for real usage. To reduce the measurement overhead, one can instrument GPU binaries with SASSI [46], NVBit [48], and GTPin [21], or bytecode using LLVM [27]. However, these instrumentation engines do not directly identify value-related inefficiencies. State-of-the-art profilers for identifying value-related inefficiencies include RedSpy [51], LoadSpy [47], and Witch [52]; however, these tools work on CPUs only. In this section, we only discuss most related approaches to ValueExpert.

Value Profiling on GPU. Xiang et al. [55] optimize instructions producing uniform values with a hardware instruction reuse buffer. Kim et al. [24] propose a hardware design to handle affine value structures. Wang and Lin [49] decouple affine value instructions from the regular SIMT instruction pipeline. Unlike these tools, ValueExpert analyzes more value patterns and requires no hardware extension.

There exist some value profilers on GPUs. Diogenes [50] overloads GPU memory copy APIs to analyze duplicate values copied to the GPU but it does not analyze patterns of value use by GPU kernels. The most related approach is GVProf [58], a value profiler for NVIDIA GPUs. While GVProf can identify value redundancies, it does not systematically categorize value patterns and cannot identify as many inefficiencies as ValueExpert can, as we describe in Section 7. Furthermore, GVProf copies measurement data from GPU to CPU for analysis, causing frequent GPU-CPU communication and prohibitively high analysis overhead for practical applications. Lastly, GVProf limits its analysis to individual GPU kernels, without insight of how values change across kernels.

Value Flow Analysis. ValueExpert's value flow analysis is a variant of data flow analysis used in GPU programming models and frameworks, such as OpenMP task dependency graph (TDG) [38, 56], CUDA Graph [30], and automatic differentiation systems in deep learning frameworks [4, 10, 39]. Unlike existing approaches, ValueExpert *dynamically* captures the data flow to guide value-related optimizations.

Tensorflow's monitoring framework—TensorBoard [17] supports a data flow view and value analysis. A fundamental limitation of TensorBoard is that it works for deep learning frameworks only and is generally applicable to other GPU-accelerated applications. In addition, ValueExpert differs from TensorBoard in several ways: (1) TensorBoard does not guide optimizations for value-related inefficiencies; (2) Unlike ValueExpert, which uses binary instrumentation, TensorBoard instruments program source code to collect graph topology and inspects tensor values; (3) TensorBoard only analyzes the distribution of values in the end of kernels/iterations but does not capture values within GPU kernels.

3 VALUE PATTERN CATEGORIZATION

In this section, we characterize eight pervasive value patterns found in GPU-accelerated applications and further categorize them into coarse- and fine-grained patterns. Table 1 overviews value patterns residing in popular Rodinia benchmarks [13] and many applications. We elaborate on each value pattern with examples.

3.1 Coarse-Grained Value Patterns

Coarse-grained value patterns describe value characteristics after each GPU API invocation. We define two coarse-grained patterns.

Definition 3.1 (Redundant Values)**.** A data object $\mathcal{D}$ matches the *redundant values* pattern at a GPU API $\mathcal{A}$ if $\mathcal{D}$ is written by A and some or all of $\mathcal{D}$'s elements are not changed by $\mathcal{A}$.

Coarse-grained value patterns are common in GPU-accelerated applications. One common cause of the redundant values pattern is double initialization of data objects — a data object may be initialized twice with the same values. In such a case, one of the initialization operations is redundant. Section 8.2 illustrates an example of this pattern found in PyTorch.

Definition 3.2 (Duplicate Values)**.** A data object $\mathcal{D}_1$ matches the *duplicate values* pattern with another data object $\mathcal{D}_2$ if $\mathcal{D}_1$ and $\mathcal{D}_2$ have the same values at any GPU API.

The duplicate values pattern occurs across GPU API invocations. For instance, in Listing 2, Darknet initializes the weight arrays of each layer on the CPU and then copies them to the GPU via memory copy APIs. These APIs copy duplicate values. One can directly invoke memory set APIs to initialize all weights on the GPU to avoid CPU-GPU memory traffic.

Table 1: Various value patterns exist in GPU applications and benchmarks.

Applications	Redundant Values	Duplicate Values	Frequent Values	Single Value	Single Zero	Heavy Type	Structured Values	Approximate Values
Rodinia/bfs [13]	✓		✓	✓		✓		
Rodinia/backprop [13]	✓	✓			✓			
Rodinia/sradv1 [13]		✓	✓	✓		✓	✓	
Rodinia/hotspot [13]			✓					✓
Rodinia/pathfinder [13]	✓		✓			✓		
Rodinia/cfd [13]	✓		✓					
Rodinia/huffman [13]	✓	✓		✓		✓		
Rodinia/lavaMD [13]	✓							
Rodinia/hotspot3D [13]								✓
Rodinia/streamcluster [13]	✓							
Darknet [44]	✓	✓	✓	✓				
QMCPACK [23]	✓							
Castro [5]	✓							
BarraCUDA [26]	✓		✓					
PytTrch-Deepwave [7]	✓			✓	✓			
PyTorch-Bert [14]	✓							
PyTorch-Resnet50 [19]	✓				✓			
NAMD [41]	✓				✓	✓		
LAMMPS [42]	✓		✓					

3.2 Fine-Grained Value Patterns

Fine-grained value patterns are identified based on *all* accesses to a data object at individual GPU APIs. We define six value patterns in this category.

Definition 3.3 (Frequent Values)**.** A data object $\mathcal{D}$ matches the *frequent values* pattern at a GPU API $\mathcal{A}$ if accesses to one or more particular values in $\mathcal{D}$ exceeds a predefined percentage threshold $\mathcal{T}$ of accesses to $\mathcal{D}$.

Definition 3.4 (Single Value)**.** A data object $\mathcal{D}$ matches the *single value* pattern at a GPU API $\mathcal{A}$ if all of $\mathcal{D}$'s accessed values are the same.

Definition 3.5 (Single Zero)**.** A data object $\mathcal{D}$ matches the *single zero* pattern at a GPU API $\mathcal{A}$ if all of $\mathcal{D}$'s accessed values are zeros.

The frequent values pattern exposes redundant computation on identical values. One can optimize it with conditional computation, which bypasses redundant computation. One example is Rodinia/huffman, where we observe that most values written to the array `histo` are zeros. To avoid identity computation, we bypass the computation on this array when zeros are found. The single value and single zero patterns are special cases of the frequent values pattern. They expose additional optimization opportunities, such as contracting a vector to a scalar to reduce memory traffic or applying a sparse data structure or algorithm to reduce computation intensity.

Definition 3.6 (Heavy Type)**.** A data object $\mathcal{D}$ matches the *heavy type* pattern at a GPU API $\mathcal{A}$ if $\mathcal{D}$'s data type is more expressive than the values used in $\mathcal{D}$.

The heavy type pattern identifies opportunities for contracting the value type to reduce memory traffic. As an example, the values in the `g_cost` array in Rodinia/bfs are always in the range of `int8` according to its input. Thus, demoting `int32` to `int8` can significantly improve the performance.

Definition 3.7 (Structured Values)**.** A data object $\mathcal{D}$ matches the *structured values* pattern at a GPU API $\mathcal{A}$ if the values accessed in $\mathcal{D}$ and the memory addresses storing these values are linearly correlated.

The structured values pattern exposes the relationship between values and the memory addresses storing these values in a data object. In other words, if the structured values pattern exists, one can infer the values stored in an array using the indices. The linear correlation between values and indices is the most common. As an example in Rodinia/srad_v1, four arrays `d_iN`, `d_iS`, `d_jW`, and `d_jE` store the coordinates of their neighbors, showing the structured value pattern. A typical optimization for this pattern is to compute the values based on the memory addresses (or array indices) to replace more costly memory load or store operations.

Definition 3.8 (Approximate Values)**.** A data object $\mathcal{D}$ matches the *approximate values* pattern at a GPU API $\mathcal{A}$ if the values accessed in $\mathcal{D}$ are *floating-point numbers* and the values with a mantissa of $\mathcal{K}$ bits correspond to some fine-grained patterns.

If approximate computing is allowed, relaxing the exact value patterns to approximate value patterns can expose more optimization opportunities. The hotspot3D code of Rodinia falls into such an example. By controlling the accuracy loss within 2% RMSE [54], one can observe the array `tIn_d` with the single value pattern and apply optimizations accordingly.

4 VALUEEXPERT OVERVIEW

ValueExpert is designed to identify the aforementioned eight value patterns via monitoring fully optimized, unmodified GPU-accelerated binaries on existing systems equipped with NVIDIA GPUs based on the Maxwell architecture or later. Figure 1 illustrates ValueExpert's major components, including performance data collection, online and offline data analysis, and a GUI for profiling results.

Data Collector. ValueExpert overloads GPU APIs, including memory copy (i.e., `cudaMemcpy` family functions), memory set (i.e., `cudaMemset` family functions), and kernel launch, to capture value snapshots (i.e., the bit-wise values of data objects) to check coarse-grained value patterns. To capture necessary information to identify fine-grained value patterns, ValueExpert utilizes NVIDIA's Sanitizer API [31] to instrument each GPU memory instruction to

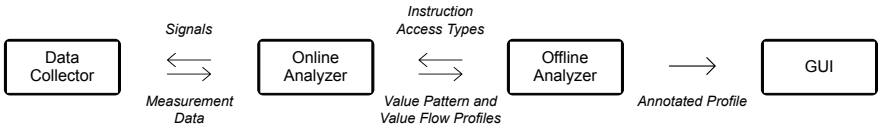


Figure 1: Overview of ValueExpert.

obtain the effective addresses of memory locations it accesses, values loaded from or stored to the addresses, and the program counter (PC) of the instruction. The data collector serializes concurrent GPU streams, maintains collected data in a pre-allocated GPU buffer, and transfers the data to the CPU when the buffer is full.

Online Analyzer. ValueExpert's online analyzer processes measurement data to identify value patterns and build value flow graphs. The online analyzer distributes the analysis work across GPUs and CPUs. For analysis that can benefit from high parallelism (e.g., parallel prefix scan), ValueExpert dispatches it to the GPU to accelerate analysis and minimize memory traffic between CPUs and GPUs. Moreover, ValueExpert captures information that is only available at runtime for deeper insights. Such information includes dynamic libraries loaded, call paths for GPU APIs, and data object allocations. Furthermore, the online analyzer works together with the offline analyzer (described in the next section) for recognizing value patterns. The output of the online analyzer consists of a profile with coarse- and fine-grained value patterns, and a program-wide value flow graph.

Offline Analyzer. ValueExpert's offline analyzer mainly analyzes CPU and GPU binaries to provide intuitive optimization guidance. ValueExpert obtains information about line mapping (i.e., source code lines, files) from the debugging sections in executables and dynamically loaded libraries, and associates them with the value pattern profile and the value flow graph. Moreover, the offline analyzer extracts the access type (i.e., value type and length) of each GPU memory instruction and provides it to the online analyzer to refine the value analysis. The type information is particularly useful to identify the heavy type pattern. The output of the offline analyzer is an annotated profile that can be visualized in a GUI.

GUI. ValueExpert provides a user-friendly GUI to visualize value patterns and flows with rich information to guide optimization. Figure 2 shows an example presentation of ValueExpert's GUI. ValueExpert GUI presents a value flow graph. The construction and annotations on this graph are elaborated in Section 5.2. The GUI quantifies coarse-grained value patterns on each vertex and edge. For each vertex, one can use its ID to look up its fine-grained value patterns in the value pattern profile. Furthermore, the GUI enables users to explore the value changes of any data object along specific paths. One can use the GUI to inspect an important portion of the graph, which is especially useful for large profiles collected from real application execution.

Figure 2 shows a part of the value flow graph produced by ValueExpert when analyzing an execution of Darknet. There are nodes with different shapes: each rectangle represents a data allocation, which is the beginning of the value flow of a data object (e.g., array); each circle represents a memory operation; each oval represents a GPU kernel. The node size is proportional to the number of invocations. The number on each node is the ID used to query its calling context and source code mapping (as shown in the black text box for vertex 220), as well as fine-grained value patterns (not shown in the figure). Edges represent value accesses: green edges denote benign value patterns, while red edges denote the redundant values pattern. The thickness of edges quantifies the number of bytes accessed. ValueExpert suggests focusing on thick red edges first for coarse-grained value patterns.

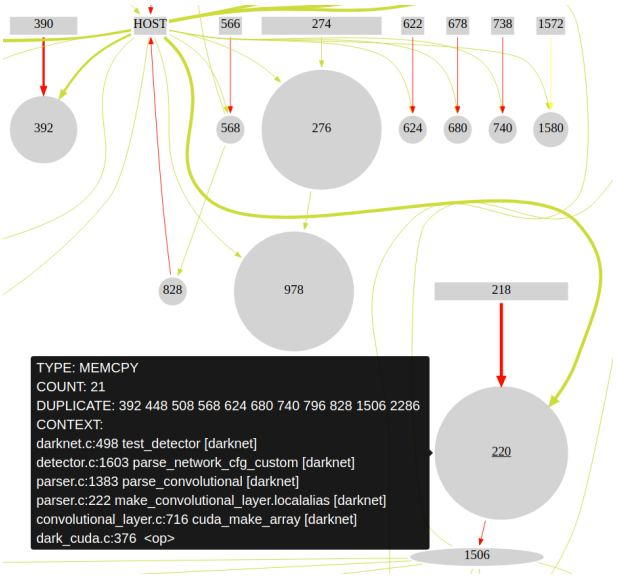


Figure 2: A part of the value flow graph for Darknet generated by ValueExpert. When a user hovers the cursor over a vertex, a text box appears to show details such as the vertex's calling context to help the user locate inefficient code.

ValueExpert's Workflow. To analyze a GPU-accelerated application, we recommend the following workflow using ValueExpert. First, enable ValueExpert's coarse-grained value pattern analysis, which generates a value flow graph with redundant values and duplicate values. From the value flow graph, users can identify costly data movement associated with GPU APIs using the important graph analysis, described in Definition 5.3. For costly data movement edges in the important graph, the user can compute a vertex slice graph, described in Definition 5.2, for GPU kernels associated with the data movement. Then, specify interesting GPU kernels (by name) to ValueExpert and enable fine-grained value pattern analysis on these kernels.

5 CORE FUNCTIONALITY IMPLEMENTATION

This section describes the implementation of ValueExpert's core functionality, including value pattern recognition and value flow analysis.

5.1 Value Pattern Recognition

ValueExpert analyzes value patterns at the level of GPU data objects. ValueExpert intercepts object allocation and deallocation functions to determine the life cycle of each data object created in GPU global memory. At each GPU memory allocation, ValueExpert records a data object's allocation context, starting address, and size. Since there is no explicit allocation function for objects on GPU shared memory, ValueExpert treats the entire shared memory as a single object. ValueExpert uses different mechanisms to identify coarse- and fine-grained value patterns.

Identifying Coarse-grained Value Patterns. Upon invocation of each GPU API $\mathcal{A}$ (i.e., GPU memory copy, memory set, and kernel launch), ValueExpert investigates each involved data object $\mathcal{D}$ by capturing its value snapshot. To recognize $\mathcal{D}$, ValueExpert overloads $\mathcal{A}$ and determines the data objects accessed by $\mathcal{A}$. A data object's value snapshot is updated upon the exit from $\mathcal{A}$ and is maintained on the CPU to reduce the GPU memory consumption. ValueExpert assesses the coarse-grained value patterns upon each value snapshot update.

Redundant values pattern: ValueExpert compares the value snapshots of a data object $\mathcal{D}$ before and after each GPU API $\mathcal{A}$ to determine the percentage of unchanged values. If the percentage is higher than a predefined threshold,[1] ValueExpert reports the redundant values pattern for $\mathcal{D}$.

Duplicate values pattern: ValueExpert calculates a `SHA256` hash [3] for the value snapshot of a data object $\mathcal{D}$ after the invocation of each GPU API $\mathcal{A}$. ValueExpert then groups data objects that have the same `SHA256` hash and reports the duplicate values pattern for these data objects.

Identifying Fine-grained Value Patterns. ValueExpert identifies fine-grained value patterns by intercepting every memory access during each GPU kernel execution. ValueExpert, leveraging NVIDIA's Sanitizer API [31], instruments callbacks at every memory load and store instructions in GPU binaries to collect each instruction's virtual program counter (PC), accessed memory address and size, and the raw value stored in this memory address. To obtain the complete information about the values accessed, ValueExpert monitors every GPU thread. ValueExpert then collects the information from all threads into a GPU buffer and copies the buffer to the CPU when it is full. This process repeats until the GPU kernel is finished.

ValueExpert then translates raw values to real values with type information because the raw values can be interpreted in different ways. For instance, a `STG.64` instruction can store either two 32-bit values or a single 64-bit value. Thus, ValueExpert analyzes the *access type* of each memory instruction, including value type (e.g., float or integer), value size (e.g., 32- or 64-bit), and number of values.

ValueExpert's offline analyzer adopts a bidirectional slicing algorithm [58] that derives a GPU memory instruction's access type based on instructions with known access types on its def-use chains. For each executed GPU memory instruction, the online analyzer transforms the instruction's virtual PC to a relative offset in a GPU binary to obtain this instruction's corresponding access type. Using information about the access types, the online analyzer interprets the raw bits of accessed values to analyze value patterns of each data object.

Challenges. There are challenges in both coarse- and fine-grained value pattern analyses. First, comparing value snapshots for a large memory range in coarse-grained analysis can incur significant overhead. Second, monitoring every GPU instruction and thread for fine-grained value pattern analysis can also incur significant overhead. Such a large overhead can limit ValueExpert from analyzing applications. Section 6 describes our optimization techniques to reduce the overhead of ValueExpert.

5.2 Value Flow Graph Construction and Analysis

Unlike existing tools that provide a profile or trace view to present performance metrics, ValueExpert constructs a *value flow graph*, which visualizes the value changes across GPU APIs to provide performance insights for optimization.

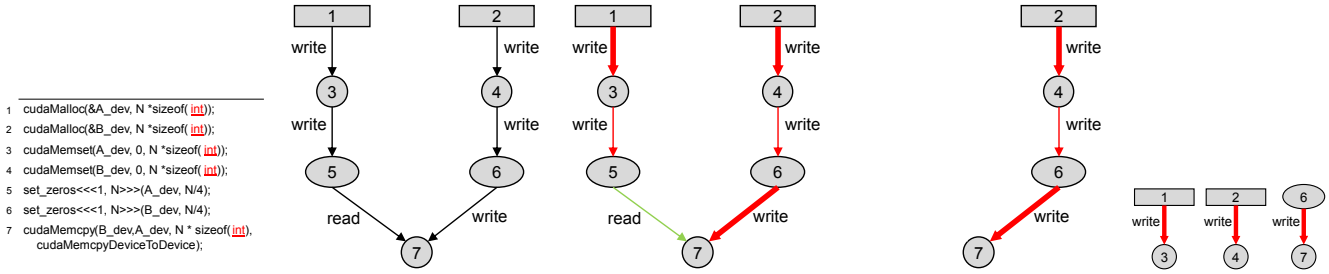
Definition 5.1 (Value Flow Graph)**.** A value flow graph $G = (V, E, v_{host})$ is a directed graph, where V is the set of vertices and E is the set of edges, and v_{host} represents any host memory operation.

- Each vertex $v \in V$ represents a GPU API invocation such as GPU memory allocation, memory copy, memory set, or kernel launch.
- An edge $e_{i,j,k} \in E$ exists from v_i to v_j if
  - v_j *writes* $\mathcal{D}_{v_k}$ or v_j *reads* $\mathcal{D}_{v_k}$, where $\mathcal{D}_{v_k}$ is a data object allocated by v_k,
  - v_i *writes* $\mathcal{D}_{v_k}$,
  - no v_u *writes* $\mathcal{D}_{v_k}$ following the write by v_i and before v_j, and
  - $e_{i,j,k}$ is labelled with read/write operations for vertex v_j.
- $e_{i,host,k}$ is a *sink* edge that represents the device to host memory transfer.
- $e_{host,i,k}$ is a *source* edge that represents the host to device memory transfer.

Figure 3 shows an example of mapping a GPU program to a value flow graph based on Definition 5.1. For convenience, we use the line number at which a GPU API is called as its ID in the value flow graph. At Lines 1 and 2, we create two vertices representing two allocated data objects. Next, at Lines 3 and 4, we create two vertices for `cudaMemset` invocations. Because Lines 3 and 4 write zeros to `A_dev` and `B_dev` respectively, we create edges from 1 to 3, and 2 to 4. Then, GPU kernels are invoked at Line 5 and Line 6 to write zeros to data object `A_dev` and `B_dev` respectively, triggering two new write edges. Finally, a read edge is created to indicate Line 7 reads data object `A_dev` from Line 5, and a write edge is created to indicate Line 7 writes data object `B_dev` from Line 6.

ValueExpert associates value patterns with value flow graphs. As shown in Figure 3, ValueExpert uses edge colors to represent redundancy and thicknesses to quantify accessed bytes. The size of each vertex is determined by an *importance factor*, which could be

[1] Based on our experiments, we use a threshold of 33%.



(a) An example program (b) A value flow graph (c) A value flow graph with value patterns. (d) A vertex slice graph focusing on node 6. (e) An important graph

Figure 3: An example of construction and analysis of a value flow graph. Rectangles are GPU memory allocations, circles are GPU memory APIs, and ovals are GPU kernels. The wider the edge, the more bytes accessed. The red color indicates high redundancy, and the green color indicates low redundancy. To facilitate the presentation, we use the line number as the ID for each vertex. Fined-grained value patterns and calling contexts (not shown) are associated with each vertex.

this API's total amount of invocations or execution time. A value flow graph is *context sensitive*. At runtime, ValueExpert records the call path of each GPU API invocation and assigns a unique ID to denote this call path. Postmortem, ValueExpert annotates the program source information for every frame on the call path as well as inline frames. Vertices with the same call path are merged to simplify presentation.

When profiling production applications such as LAMMPS [42], ValueExpert can generate a huge value flow graph. To facilitate the analysis, we describe two features that can help one explore interesting subgraphs.

Definition 5.2 (Vertex Slice Graph)**.** A vertex slice graph $G_B(v_u) = (V', E', v_{host})$ is a subgraph of a value flow graph $G = (V, E, v_{host})$ where

- $e_{i,j,k} \in E'$ if $e_{i,j,k} \in E$ and
  - v_u *writes* $\mathcal{D}_{v_k}$ or v_u *reads* $\mathcal{D}_{v_k}$, and
  - $e_{i,j,k}$ is on a valid path that consists of edges that *read* or *write* $\mathcal{D}_{v_k}$ and reaches v_u or v_u reaches.
- $v \in V'$ if v is on any edge $e \in E'$.

Figure 3d shows that applying vertex slice analysis according to Definition 5.2 on vertex 6 generates a *vertex slice graph* $G_B(v_6)$ that tracks vertex 6's inputs and outputs. Vertices that do not affect vertex 6's value patterns and vertices whose value patterns are not affected by vertex 6 are eliminated.

We use $I(x)$ to represent user-defined metrics that measure the importance of a vertex or an edge. $\mathcal{I}_e$ is the threshold for keeping an edge in a graph, and $\mathcal{I}_v$ is the threshold for keeping a vertex in a graph. We define *important graph* using Definition 5.3.

Definition 5.3 (Important Graph)**.** An important graph $G_I = (V', E', v_{host})$ is a subgraph of $G = (V, E, v_{host})$ where

- $e_{i,j,k} \in E'$ if $e_{i,j,k} \in E$ and $I(e_{i,j,k}) \geq \mathcal{I}_e$
- $v \in V'$ if v is on any edge $e \in E'$ or $I(v) \geq \mathcal{I}_v$

We let $I(e)$ be accessed bytes on each edge, and $I(v)$ be the number of invocations of the GPU API represented by each vertex. With $\mathcal{I}_e = N/2$ and $\mathcal{I}_v = 1$, we can prune the graph in Figure 3d and yield the graph in Figure 3e with important vertices and edges only. Applying the important graph analysis, ValueExpert trims the original value flow graph of LAMMPS from 660 nodes and 1258 edges to 132 nodes and 97 edges.

6 ACCELERATING VALUE ANALYSIS

Without optimization, ValueExpert's value pattern analysis can incur unaffordable overhead due to the heaveweight instrumentation and frequent GPU-CPU communication. For example, without any optimization, ValueExpert slows down Rodinia/streamcluster by 1200×, and it does not even finish the measurement of complex applications, such as LAMMPS and PyTorch. Thus, we adopt several optimizations to reduce ValueExpert's overhead for both coarse- and fine-grained value pattern analysis.

6.1 Accelerating Coarse-Grained Analysis

Problem statement. We compare the value snapshots $\mathcal{V}(\mathcal{D})_{\mathcal{A}}$ and $\mathcal{V}'(\mathcal{D})_{\mathcal{A}}$ of a data object $\mathcal{D}$ at a GPU API $\mathcal{A}$. $\mathcal{V}(\mathcal{D})_{\mathcal{A}}$ is the snapshot before executing $\mathcal{A}$, and $\mathcal{V}'(\mathcal{D})_{\mathcal{A}}$ is the snapshot after executing $\mathcal{A}$. As described in Section 5.1, comparing the value snapshots involves substantial computations. To understand the redundant values pattern and avoid unnecessary comparisons over a large memory range, ValueExpert only compares the values stored in memory addresses that are accessed by $\mathcal{A}$. If the portion of a data object that is never accessed by $\mathcal{A}$ is large, ValueExpert suggests avoiding unnecessary GPU data allocation. Otherwise, ValueExpert suggests investigating the use of values.

We define an interval [`start`, `end`] as the memory range accessed by each GPU instruction in $\mathcal{A}$. As a GPU kernel executes, a vast number of intervals can be generated. ValueExpert merges these intervals if they are adjacent or overlapped and copies values after interval merging to the CPU for efficient processing.

Solution – Employing GPU Parallelism. One could copy all intervals from the GPU to the CPU and perform a sequential interval merge, which has a $O(NlogN)$ complexity, where N denotes the number of intervals. This algorithm, however, only works for small GPU kernels as N can be large in many benchmarks and applications (e.g., 3.4×10^7 for each kernel in *streamcluster*), triggering large

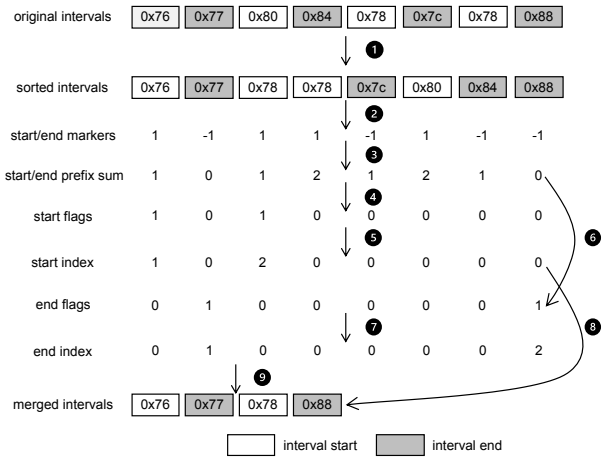


Figure 4: An example of merging intervals in parallel.

CPU processing and GPU-CPU memory copy overhead. To merge intervals on the GPU, we develop the parallel interval algorithm shown in Figure 4.

❶ We first lexicographically sort the start and end addresses of all intervals based on ⟨*address*, *is_end*⟩ pairs such that an end address is after a start address when they are equal. ❷ Next, we initialize a *markers* array to denote interval starts with 1 and interval ends with -1. ❸ We apply a parallel prefix scan on the markers array. The merged intervals cover a number of original intervals such that the prefix sum of merged interval starts are 1 and the interval ends are 0. ❹ we create a *start flags* array. Each entry in the array is zero, unless the corresponding start/end prefix sum value is 1 and the entry represents an interval start. ❺ We apply another parallel prefix scan to get output indices of the merged interval starts. The output indices of the merged interval ends are obtained similarly through steps ❻ and ❼. Finally, we place the merged interval starts and interval ends to the output buffer (❽ and ❾). The complexity of this parallel interval merge algorithm is $O(logN)$ using parallel radix sort. This algorithm is further optimized to merge the intervals accessed by threads within the same warp using efficient warp primitives (i.e., `shfl`, `bfe`, `bfind`, and `brev`). We refer to this simplified version as *interval compaction*.

We implement this parallel interval merging algorithm in ValueExpert as a concurrent data processing GPU kernel that launched before each application kernel to merge intervals on-the-fly. Based on the most room policy [16], we let the data processing kernel occupy all resources of some streaming multiprocessors [2] so that it won't be slowed down by sharing resources with the application kernel. The application kernel keeps putting accessed intervals in a GPU buffer. Once this buffer is full, the data processing kernel applies the interval compaction algorithm to merge the intervals within the same warp. After the compaction is done, the data processing kernel notifies the application kernel and lets it resume execution. The data processing kernel then applies the parallel interval merging algorithm described in Figure 4 while the application kernel is executing. When the application kernel ends,

[2]We use one GPU block in all experiments.

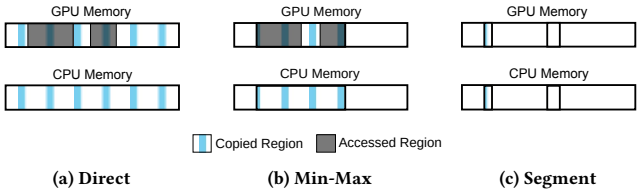


Figure 5: Three memory copy strategies in ValueExpert.

merged intervals will be copied to the CPU. Compared with existing instrumentation-based GPU tools [40, 58] that perform data processing at instrumentation callbacks or on CPUs, our approach employs highly-parallel data processing using many GPU threads, little CPU-GPU transfer time, and concurrent execution of data processing and application kernels.

Solution – Minimizing CPU-GPU Data Transfers. After obtaining the merged intervals, ValueExpert updates each involved data object's value snapshot on CPU. On the one hand, if we copy the entire interval of each data object (i.e., direct copy in Figure 5a), we waste time copying values that are not accessed. On the other hand, if we only copy values on the accessed addresses (i.e., segment copy in Figure 5c), we need invoke to memory copy APIs many times. ValueExpert employs an adaptive copy mechanism to switch between different copy strategies. Besides the aforementioned two strategies, ValueExpert supports a third one—min-max copy (i.e., Figure 5b), which copies memory based on the minimum and maximum addresses across all intervals.

ValueExpert employs the segment copy when the distribution of accessed intervals is sparse and the number of intervals is small, and switches to the min-max copy when the distribution is dense or the number of intervals is large.

6.2 Accelerating Fine-Grained Analysis

Filtering. ValueExpert supports monitoring a subset of GPU kernels specified by users. One can easily use domain knowledge or another profiling pass to identify important or hot GPU kernels for fine-grained value pattern analysis.

Sampling. Often, GPU kernels show similar behaviors across loop iterations and across GPU thread blocks, such that their value patterns can be identified with sampled kernels and blocks. Based on the observation, we adopt a hierarchy sampling method [58] to sample GPU kernels and blocks to further reduce overhead.

7 EVALUATION

We evaluate ValueExpert on two platforms equipped with NVIDIA GPUs shown in Table 2. We use ValueExpert to analyze nine applications and the Rodinia benchmark suite [13]. We study these applications because Darknet and Pytorch are popular deep learning frameworks and the rest are important HPC applications.

Speedups. We had a graduate student optimize GPU applications guided by ValueExpert. This student was familiar with ValueExpert but had no prior knowledge about the applications. Typically, this student spent a few hours to apply and verify an

Table 2: The configurations of two GPU systems to evaluate ValueExpert: RTX 2080 Ti and A100 GPUs.

CPU	GPU	GPU Multiple-processors	GPU Memory Size	System	GPU Driver	CUDA Toolkit	GCC	Compiler Options
Intel(R) Xeon(R) 6226	RTX 2080 Ti	72	11GB GDDR6	Linux 5.4.0	460.27	11.1.1	9.3.0	`-g -lineinfo -O3 -arch sm_75`
AMD EPYC 7402	A100	108	40GB HBM2	Linux 4.18.0	460.27	11.1.1	8.3.1	`-g -lineinfo -O3 -arch sm_80`

Table 3: Evaluation of kernel execution time, memory time (i.e., memory allocation, copy, and set), and corresponding speedups for Rodinia benchmarks and some real applications on RTX 2080 Ti and A100 GPUs. For deep learning applications, including Darknet, Deepwave, Bert, and Resnet50, we report layer-level speedups because our optimizations improve multiple GPU kernels. For streamcluster, QMCPACK, and LAMMPS, we do not report kernel speedups because our optimizations accelerate memory operations only. We also report standard deviations of the speedups indicated by ± in the table.

Application	Kerne Name	RTX 2080 Ti				A100			
		Kernel Time	Speedup	Memory Time	Speedup	Kernel Time	Speedup	Memory Time	Speedup
Rodinia/bfs [13]	Kernel	939.12us	1.34× ±0.00	13.43ms	1.10× ±0.01	500.29us	0.99× ±0.00	3.17ms	1.20× ±0.05
Rodinia/backprop [13]	bpnn_adjust_weights_cuda	106.41us	8.18× ±0.00	2.80ms	1.01× ±0.01	17.72us	1.67× ±0.02	1.36ms	1.01× ±0.01
Rodinia/sradv1 [13]	srad	198.60us	1.52× ±0.00	175.15us	1.03× ±0.01	89.65us	1.11× ±0.00	183.11us	1.06× ±0.00
Rodinia/hotspot [13]	calculate_temp	79.78us	1.31× ±0.00	257.43us	1.00× ±0.00	19.24us	1.10× ±0.00	195.22us	1.00× ±0.00
Rodinia/pathfinder [13]	dynproc_kernel	134.00us	1.13× ±0.01	15.66ms	4.21× ±0.56	153.36us	1.37× ±0.00	2.68ms	3.27× ±0.13
Rodinia/cfd [13]	cuda_compute_flux	296.55us	8.28× ±0.04	1.10ms	1.01× ±0.00	212.98us	6.05× ±0.01	613.00us	1.03× ±0.01
Rodinia/huffman [13]	histo_kernel	91.35us	1.49× ±0.04	346.11us	1.00× ±0.00	240.13us	2.55× ±0.10	392.25us	1.00× ±0.00
Rodinia/lavaMD [13]	kernel_gpu_cuda	62.93ms	0.99× ±0.01	1.70ms	1.49× ±0.02	4.40ms	0.98× ±0.00	822.90us	1.39× ±0.01
Rodinia/hotspot3D [13]	hotspotOpt1	4.80ms	2.00× ±0.00	8.81ms	1.00× ±0.01	2.20ms	1.99× ±0.01	2.68ms	0.99× ±0.02
Rodinia/streamcluster [13]	-	-	-	723.92ms	2.39× ±0.03	-	-	422.68ms	1.81× ±0.08
Darknet [44]	convolution	1.91ms	1.06× ±0.00	21.84ms	1.82× ±0.01	2.45ms	1.05× ±0.01	10.66ms	1.73× ±0.04
QMCPACK [23]	-	-	-	4.13ms	1.00× ±0.00	-	-	11.02ms	1.00× ±0.00
Castro [5]	cellconslin_slopes_mmlim	21.28ms	1.27× ±0.00	141.06ms	1.00× ±0.02	25.55ms	1.24× ±0.01	131.47ms	1.02× ±0.02
BarraCUDA [26]	cuda_inexact_match_caller	58.01ms	1.06× ±0.01	41.50ms	1.13× ±0.01	32.06ms	1.06× ±0.00	18.45ms	1.13× ±0.02
PyTorch-Deepwave [7]	ReplicationPad	1.63ms	1.07× ±0.01	15.99us	1.01× ±0.01	1.24ms	1.04× ±0.01	45.26us	1.00× ±0.00
PyTorch-Bert [14]	embedding	9.77ms	1.57× ±0.05	59.28us	1.01× ±0.01	11.77ms	1.59× ±0.04	62.98us	1.00× ±0.01
PyTorch-Resnet50 [19]	convolution	164.70ms	1.02× ±0.01	16.66ms	1.00× ±0.03	241.48ms	1.03× ±0.02	23.60ms	0.98× ±0.02
NAMD [41]	nonbondedForceKernel	22.50ms	1.00× ±0.00	16.43ms	1.00× ±0.01	34.50ms	1.00× ±0.01	34.84ms	1.00× ±0.01
LAMMPS [42]	-	-	-	50.76ms	6.03× ±0.01	-	-	56.72ms	5.19× ±0.32
Geometric Mean	-	-	1.58×	-	1.34×	-	1.39×	-	1.28×
Median	-	-	1.29×	-	1.01×	-	1.11×	-	1.02×

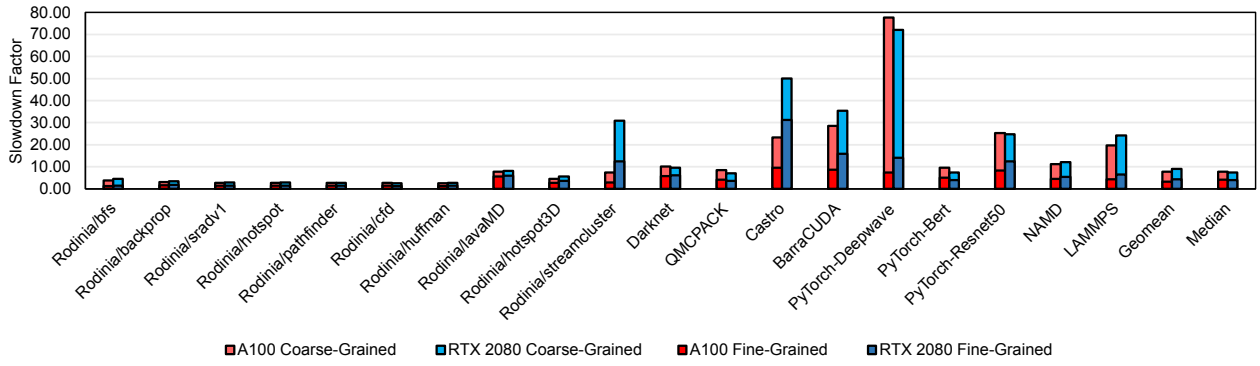


Figure 6: Evaluation of ValueExpert's fine- and coarse-grained overhead on benchmarks and applications. ValueExpert does not use any sampling technique for profiling coarse-grained value patterns. In fine-grained analysis, we set block and kernel sampling periods as 20 for benchmarks, and 100 for applications; we monitor all GPU kernels for benchmarks without kernel filtering and monitor one of the hottest kernels with kernel filtering for each application.

optimization. As shown in Table 3, that student obtained nontrivial performance improvements for programs running on both RTX 2080 Ti and A100 under the guidance of ValueExpert. We report the execution time of optimized GPU kernels and all memory operations (memory allocation, set, and copy) obtained from NVIDIA Nsight Systems [36] and their corresponding speedups. We observe that our optimizations typically yield a higher speedup on RTX 2080 Ti than A100 because A100's HBM2 has a bandwidth higher than that of RTX 2028 Ti's GDDR6; thus, reducing memory load and store operations can improve performance more on RTX 2080

Table 4: Benchmark speedups obtained by leveraging one or more value patterns.

Application	Pattern	RTX 2080Ti		A100	
		Kernel Speedup	Memory Speedup	Kernel Speedup	Memory Speedup
backprop	Single Zeros	8.18×	0.99×	1.67×	1.20×
	Duplicate Values	1.00×	1.00×	1.00×	1.00×
bfs	Heavy Type	1.34×	1.08×	0.97×	0.99×
	Frequent Values	1.00×	1.10×	1.01×	1.01×
pathfinder	Heavy Type	1.13×	4.21×	1.37×	3.27×
sradv1	Heavy Type	1.40×	1.00×	1.05×	1.02×
	Structured Values	1.05×	1.02×	1.08×	1.07×
hotspot	Approximate Values	1.31×	1.00×	1.10×	1.00×
cfd	Frequent Values	8.25×	1.00×	6.06×	1.02×
	Redundant Values	1.00×	1.02×	1.00×	1.00×
hotspot3d	Approximate Values	2.00×	1.00×	1.99×	0.99×
streamcluster	Redundant Values	-	2.39×	-	1.48×
huffman	Frequent Values	1.49×	1.00×	2.55×	1.00×
lavaMD	Heavy Type	0.99×	1.49×	0.98×	1.39×
Darknet	Redundant Values	1.06×	1.82×	1.05×	1.73×
QMCPACK	Redundan Values	-	1.00×	-	1.00×
Castro	Redundan Values	1.27×	1.00×	1.24×	1.02×
BarraCUDA	Redundan Values	1.06×	1.13×	1.06×	1.13×
PytTrch-Deepwave	Redundant Values	1.07×	1.01×	1.04×	1.33×
PyTorch-Bert	Redundant Values	1.57×	1.01×	1.59×	1.00×
PyTorch-Resnet50	Single Values	1.02×	1.00×	1.03×	0.98×
NAMD	Single Zero	1.00×	1.00×	1.00×	1.00×
LAMMPS	Frequent Values	-	6.03×	-	5.19×

Ti than A100. Section 8 details some case studies. Table 4 shows the benchmark speedups obtained by leveraging one or more value patterns. We observe that the redundant values pattern is the most common pattern and optimizing the single zeros and frequent values patterns yields the most speedups.

Overhead. Figure 6 shows that ValueExpert incurs moderate overhead for both coarse- and fine-grained value pattern analysis, thanks to the optimization techniques described in Section 6. It has a median overhead of 7.35× on RTX 2080 Ti and 7.81× on A100. For coarse-grained analysis, ValueExpert has a median overhead of 3.38× and 4.28×, and a geometric mean overhead of 4.38× and 4.22× on RTX 2080 Ti and A100 respectively. For fine-grained analysis, ValueExpert has a median overhead of 3.97× and 4.18×, and a geometric mean overhead of 4.32× and 3.23× on RTX 2080 Ti and A100 respectively. A100 has a lower overhead for applications that involve significant memory accesses, including Rodinia/streamcluster, Castro, BarraCUDA, and LAMMPS, because of its higher bandwidth memory than RTX 2080 Ti. PyTorch-deepwave suffers from the highest overhead on both GPUs. This program accesses millions of different memory addresses for each kernel and results in about 100 thousand non-adjacent intervals after merging.

ValueExpert vs. GVProf. Compared to GVProf, ValueExpert enlarges the analysis scope, provides additional insights, and dramatically lowers the measurement overhead. First, GVProf cannot guide users to all of the optimizations shown in Table 3 for codes such as Castro, PyTorch, QMCPACK, and others because it limits the analysis to individual GPU kernels. Second, GVProf does not analyze values patterns or value flows. Though GVProf can identify redundant values in LAMMPS, it does not directly pinpoint GPU APIs that cause the redundancy. Third, GVProf incurs much higher overhead than ValueExpert, especially for real applications. GVProf cannot finish profiling Castro and NAMD within one day on RTX 2080 Ti, while ValueExpert finishes within five minutes.

```
1void replication_pad3d_backward_out_cuda_template(...){
2  gradInput.resize_as_(input);
3  gradInput.zero_();
4  ...}
5Tensor replication_pad3d_backward_cuda(...){
6- auto gradInput = at::zeros_like(input,
       LEGACY_CONTIGUOUS_MEMORY_FORMAT);
7+ auto gradInput = at::empty_like(input,
       LEGACY_CONTIGUOUS_MEMORY_FORMAT);
8  replication_pad3d_backward_out_cuda_template(gradInput,
       gradOutput, input, paddingSize);
9  ...
10}
```

Listing 3: The redundant values and single zero patterns in Deepwave [7]. `gradientInput` is initialized to zeros by `at::zeros_like` and the following `gradInput.zero_()`.

8 CASE STUDIES

In this section, we describe several case studies in detail. It is worth noting that our application optimizations do not introduce any accuracy loss.

8.1 Darknet

We run Darknet [44] using a pre-trained `yolov4.weights` to detect objects in *dog.jpg*. ValueExpert detects both inefficiencies described in Section 1.1. The value flow figure has 70 nodes and 114 edges. Figure 2 highlights two severe redundant value flows: 390 → 392 and 218 → 220 → 1506, which pinpoint the data objects and memory operations that trigger Inefficiency I and II, respectively. By applying optimizations described in Section 1.1, we obtain 1.06× and 1.05× speedups for convolution layers on RTX 2080 Ti and A100 correspondingly.

8.2 PyTorch

We study three neural networks written in PyTorch [39]: Deepwave, Resnet50, and Bert.

Deepwave. Deepwave [7] provides efficient wave propagation modules that perform seismic imaging/inversion. ValueExpert first reports 100% memory accesses in function `replication_pad3d_backward_cuda` matches the redundant values pattern. Listing 3 shows the problematic code. ValueExpert also highlights that the `input` tensor at Line 6 matches the single zero value pattern. `input` is allocated and initialized to zeros at Line 6 and reinitialized again at Line 3 without being accessed in between. The value flow figure has 38 nodes and 49 edges. To optimize the code, we replace the `zeros_like` function with the `empty_like` function that allocates memory without initialization at Line 7. ValueExpert reports two other tensors in `replication_pad2d_backward_cuda` and `replication_pad1d_backward_cuda` suffering from the same problem. By optimizing all of them, we obtain 1.07× and 1.04× speedups in the backward phase of the ReplicationPad operator on RTX 2080 Ti and A100, respectively. This optimization has been upstreamed to the PyTorch repository.

Resnet50. Resnet50 [19] is a 50-layer convolutional neural network. We profile its inference phase using ValueExpert to generate both coarse- and fine-grained value pattern reports for the GPU kernels in Listing 4. The value flow figure has 75 nodes and 223 edges.

```
1void THNN_(SpatialConvolutionMM_updateOutput)(...) {
2+ if (bias) {
3    if (ones->dim() != 2 || ones->size(0)*ones->size(1) <
         outputHeight*outputWidth) {
4     THCTensor_(resize2d)(state, ones, outputHeight, outputWidth);
5     THCTensor_(fill)(state, ones, ScalarConvert<int, scalar_t>::
          to(1)); }
6+ }
7}
```

Listing 4: The redundant values and single value patterns in Resnet50 [19]. The array ones is resized and initialized to zeros even it is not used later.

ValueExpert reports 14.25MB memory bytes at Line 5 involve redundant values; moreover, ValueExpert reports the single value pattern for the ones tensor. The forward phase of convolution computation can be viewed as $input \times filter + bias$. However, Resnet's convolution operators foregoes the $+bias$ calculation because its batchnorm operators that follow each convolution operator have already considered bias. Since the ones tensor is only used for accumulating bias, we can omit its allocation and initialization if bias is ignored. Simply adding the two lines eliminates the redundancy and yields 1.02× and 1.03× speedups for convolution layers on RTX 2080 Ti and A100, respectively. This patch has been upstreamed to the PyTorch repository.

Bert. Bert [14] is a transformer-based neural network for natural language processing. ValueExpert reports the out array in the embedding operator matches the redundant value pattern and 2.8KB bytes are involved. The value flow graph has 101 nodes and 217 edges. With the value flow analysis, ValueExpert shows that paddings of out is initialized to zeros in the reset_parameters function, while they are reinitialized in every call to the embedding.maskek_fill_ function in each iteration. Thus, ValueExpert suggests removing the second initialization, which yields 1.57× and 1.59× speedups for the embedding operator running on RTX 2080 Ti and A100, respectively. This issue has been confirmed by PyTorch developers.

8.3 Castro

Castro [5] is an astrophysical radiation hydrodynamics simulation code based on the AMReX framework [57]. Castro is an important application supported by Department of Energy's exascale computing project. We study Castro's Sedov example using the inputs.2d.cyl_in_cartcoords input. The value flow graph generated by ValueExpert has 1092 nodes and 1666 edges. ValueExpert reports that the array slopes matches the redundant values pattern in the GPU kernel cellconslin_slopes_mmlim, which is a function provided by AMReX. Listing 5 shows this GPU kernel, which computes a multi-dimensional array slopes. We observe that the scalar a at Line 7 is often 1.0, resulting in identity computation and unchanged values in slope. Thus, we conditionally bypass the computation when a is 1.0, which yields 1.27× and 1.24× speedups for this GPU kernel running on RTX 2080 Ti and A100, respectively. It is worth noting that this optimization improves a library function of AMReX and could benefit all its applications, not limited to Castro. This optimization has been confirmed by Castro developers.

```
1AMREX_GPU_HOST_DEVICE AMREX_FORCE_INLINE void
      cellconslin_slopes_mmlim(){
2  for (int n = 0; n < ncomp; ++n) {
3    for (int j = lo.y; j <= hi.y; ++j) {
4      for (int i = lo.x; i <= hi.x; ++i) {
5+       if (a != 1.0) {
6          slopes(i,j,0,n) *= a;
7          slopes(i,j,0,n+ncomp) *= a;
8+       }
9        ...
10}
```

Listing 5: The redundant values pattern in Castro [5]. The variable a is mostly 1.0 in our experiment of input inputs.2d.cyl_in_cartcoords. By adding the condition check at Line 5, we can save memory loads and stores.

8.4 BarraCUDA

BarraCUDA [26] is a fast sequence mapping software to map sequencing reads to a particular location on a reference genome. We study BarraCUDA using a typical input[3]. The value flow graph generated by ValueExpert has 30 nodes and 42 edges. ValueExpert reports the redundant values pattern on array global_sequences_index in function copy_sequences_to_cuda_memory. BarraCUDA invokes memory copy APIs to copy values from the CPU to the GPU for this array even when it is empty. By adding a size check, we avoid copying empty arrays and other arrays that only need to be updated when array global_sequences_index is changed. ValueExpert also reports the frequent values pattern with 99.6% zeros in array global_alns in GPU kernel cuda_inexact_match_caller. This array is copied from a thread-local array on the GPU. We create a hits array to record positions that have been updated with nonzero values, and only copy these values. With these optimizations, we obtain a 1.06× kernel execution time speedup and a 1.13× memory time speedup on both RTX 2080 Ti and A100.

8.5 Rodinia CFD and Backprop

CFD and Backprop are two Rodinia benchmarks. We study CFD using fvcorr.domn.097K input. ValueExpert reports that the kernel cuda_compute_flux has frequent values pattern on array variables. We observe that this array is initialized with values within a small range and is unchanged in the first three iterations. Thus, we hash the accessing index of this array to limit memory accesses to certain addresses, which greatly increases the data locality. This optimization yields 8.28× and 6.05× speedups on RTX 2080 Ti and A100, respectively.

We study Backprop with its built-in input. ValueExpert reports that the kernel bpnn_adjust_weights_cuda has single zeros pattern on arrays w and oldw. We conditionally bypass floating point computations and writes to these two arrays when they zeros. In this way, we obtain 8.18× and 1.67× speedups on RTX 2080 Ti and A100, respectively. RTX 2080 Ti achieves a much higher speedup because reducing FP64 can alleviate significant computation workload on this architecture with fewer FP64 units than A100.

[3] Saccharomyces_cerevisiae.SGD1.01.50.dna_rm.toplevel.fa

Table 5: ValueExpert vs. existing redundancy analysis tools.

	ValueExpert	GVProf [58]	Witch [52]	RedSpy [51]	LoadSpy [47]	RVN [53]
Redundancy analysis	Support	Support	Support	Support	Support	Support
Value pattern analysis of data objects	Support	N/A	N/A	N/A	N/A	N/A
Result granularity	GPU API	Instruction	Instruction	Instruction	Instruction	Instruction
Value flows	Support	N/A	N/A	N/A	N/A	N/A
GPU program analysis	Support	Support	N/A	N/A	N/A	N/A
Geomean overhead*	7.8×	47.3×	2.1×	19.1×	26.0×	33.9×

* Because ValueExpert, Witch, and LoadSpy profile applications using multiple runs, we sum up the overheads from all required runs.

8.6 Optimization Tradeoffs

It is worth noting that our optimizations do not always yield speedups on GPU kernels. For example, we obtain a negligible speedup on GPU kernels of Rodinia/lavaMD. For this benchmark, ValueExpert reports the heavy type pattern on array `rA`, whose elements are ten values from $\{0.1, 0.2, ..., 1.0\}$. Our optimization demotes the type from `double` to `unit8_t` and reverts it to `double` when the array is copied to the GPU. The optimization increases the GPU kernel execution time by 2% but reduces the CPU-GPU memory transfer time by 28%. For NAMD and QMCPACK, ValueExpert reports the redundant values pattern for both, and the heavy type pattern for NAMD. Our optimizations do not yield significant speedups on RTX 2080 Ti and A100 GPUs because the inefficiencies do not occur at bottleneck functions for the given inputs. Since all inefficiencies of these two applications reside in a loop nest whose trip counts depend on input, our optimizations are going to benefit other inputs that stress on these loops.

9 CONCLUSIONS

Our studies have shown that many GPU applications have value-related inefficiencies. To guide optimizations of valued-related inefficiencies, we develop ValueExpert—the first tool infrastructure that identifies and categorizes inefficient value patterns with insightful value flow graphs. To accelerate ValueExpert and make value analysis possible for non-trivial programs, we devise a data-parallel interval merge algorithm on GPUs to identify accessed data regions and only transfer accessed data from the GPU to the CPU. Table 5 compares ValueExpert with existing redundant analysis tools; ValueExpert has the following unique features:

(1) ValueExpert analyzes the value patterns of each data object. While other tools identify spatial and temporal value redundancies for individual instructions, they do not categorize value patterns.
(2) ValueExpert profiles and analyzes value inefficiencies at each GPU API, while other tools focus on individual instructions. Identifying value access patterns at APIs provides actionable optimization opportunities.
(3) ValueExpert provides unique value flow graphs to pinpoint the causes of inefficiencies and guide optimization.
(4) ValueExpert leverages GPU parallelism to accelerate some important analysis, so it incurs much lower overhead than other fine-grained GPU profilers, e.g., GVProf.

Guided by ValueExpert's performance reports with rich program context and value flow information, we are able to quickly identify the causes of value-related inefficiencies in large applications. We optimize a number of applications and well-known benchmarks, achieving nontrivial speedups. Many optimization patches have been confirmed or accepted by the developers.

Currently, ValueExpert collects program context using the line mapping section in binaries, which does not directly provide straightforward information for interpreted languages such as Python. We plan to add more semantic information into ValueExpert's performance reports to facilitate optimizations. For instance, we can integrate the layer/operator annotations in deep learning applications using source code instrumentation. Inspired by ValueExpert's fast interval merge implementation on GPUs, we intend to offload other important program analyses, such as reuse distance and race detection, to GPUs to lower the overhead for complex applications.

ACKNOWLEDGEMENT

We thank anonymous reviewers for their valuable comments, and Aurelien Chartier at NVIDIA for providing suggestions on using Sanitizer API. This research was supported in part by NSF CNS-2125813, the Oak Ridge National Laboratory Joint Faculty Appointment Grant, and the Exascale Computing Project (17-SC-20-SC) — a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact includes ValueExpert and benchmark code in this paper, along with instructions to use benchmarks to generate results for Figure 2, Figure 6, and Table 3 on NVIDIA A100 and RTX 2080 Ti GPUs. The speedup and overhead of each benchmark are averaged among 10 runs.

We provide a docker image with pre-installed prerequisites to simplify the experiment workflow. Users can also use a script to install all software from scratch.

A.2 Artifact Check-list (Meta-information)

- **Algorithm: Value flow graph construction and analysis. Parallel interval merge on GPUs.**
- **Program: Rodinia benchmark v3.1, QMCPACK@47406206, Castro@5e0a1b9c, AlexeyAB/darknet@312fd2e9,**

NAMD@4a41c608, BarraCUDA@0.7.107h, PyTorch-Deepwave@1154692258, PyTorch-Bert@f5788898, PyTorch-Resnet@f5788898
- **Compilation: GCC ≥ 7.3.1, NVCC ≥ 11.1.1**
- **Run-time environment: Provided binaries are for Linux x86-64 systems.**
- **Hardware: NVIDIA Volta GPUs and later generations.**
- **Metrics: Optimization speedups and profiling overhead**
- **Output: An example value flow graph. A file that contains speedups for all benchmarks. A file that contains profiling overhead for all benchmarks.**
- **How much disk space required (approximately)?: 200GB**
- **How much time is needed to prepare workflow (approximately)?: 2 hours**
- **How much time is needed to complete experiments (approximately)?: 4 hours**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: BSD-3**
- **Archived (provide DOI)?: https://zenodo.org/record/5796083**

A.3 Description

A.3.1 How to Access. Our benchmarks, source code, scripts are available at https://github.com/GVProf/GVProf.

A.3.2 Hardware Dependencies. ValueExpert currently only works on NVIDIA Volta GPU and generations above. We have tested ValueExpert's correctness and performance on machines equipped with NVIDIA A100 and RTX 2080 Ti GPUs. To reproduce the results in the paper, we suggest the reviewers use the same GPUs and a machine with at least 200GB available disk space.

A.3.3 Software Dependencies.
- NVIDIA CUDA driver: ≥ 460.27 and ≤ 470.57
- CUDA Toolkit: 11.1.1 and above
- GCC: 8.3.1 and above
- Linux Kernel: 4.18.0 and above
- All the other dependent software can be installed by our automate scripts.

A.4 Installation

Decompress the packages and launch a docker instance.

```
7za x ./asplos_ae_home.7z
7za x ./value_expert_ae_image.tar.7z
docker load -i ./value_expert_ae_image.tar
docker run --runtime=nvidia --rm --name=asplos_ae -t \
-v `pwd`/asplos_ae_home:/root -i \
value_expert_ae /bin/bash
```

Install ValueExpert and benchmarks.

```
cd root/GVProf/
source env.sh
# [gpu_arch]=80 if you are using A100
# [gpu_arch]=75 if you are using RTX 2080 Ti
./install.sh [gpu_arch]
```

A.5 Experiment Workflow

Reproduce overhead in Figure 6 (2 hours).

```
# [gpu_arch]=80 if you are using A100
# [gpu_arch]=75 if you are using RTX 2080 Ti
./scripts/overhead.sh [gpu_arch]
cat ./overhead.txt
```

Reproduce speedups in Table 3 (1 hour).

```
# [gpu_arch]=80 if you are using A100
# [gpu_arch]=75 if you are using RTX 2080 Ti
./scripts/speedup.sh [gpu_arch]
cat ./kernel_speedup.txt # kernel execution time speedups
cat ./mem_speedup.txt # memory operation time speedups
```

Reproduce data flow in Figure 2 (5 minutes). It will generate a demo.svg file under /root/GVProf/jquery.graphviz.svg/.

```
./scripts/figure2.sh
```

View the data flow graph in a browser. Users can access http://docker_ip:8000 in the host browser to check the data_flow graph

```
cd /root/GVProf/jquery.graphviz.svg
python3 -m http.server
```

A.6 Evaluation and Expected Results

We expect the reproduced results for Figure 2, Table 3, and Figure 6 match the results in the paper. Figure 2 shows a value flow graph for Darknet generated by ValueExpert. Table 3 shows the evaluation of kernel execution time, memory time, and corresponding speedups for Rodinia benchmarks and some real applications on RTX 2080 Ti and A100 GPUs. Figure 6 shows that ValueExpert incurs moderate overhead for both coarse- and fine-grained value pattern analysis.

Our PRs for PyTorch 48540 and 48890 have been merged by PyTorch developers.

REFERENCES

[1] 2020. TOP500. https://www.top500.org/lists/top500/2021/06/. [Accessed August 4, 2021].
[2] 2021. ROC-profiler. https://github.com/ROCm-Developer-Tools/rocprofiler. [Accessed Aug 9, 2021].
[3] 2021. SHA-2. https://en.wikipedia.org/wiki/SHA-2 [Accessed Apr 9, 2021].
[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.
[5] Ann Almgren, Maria Barrios Sazo, John Bell, Alice Harpole, Max Katz, Jean Sexton, Donald Willcox, Weiqun Zhang, and Michael Zingale. 2020. CASTRO: A Massively Parallel Compressible Astrophysics Simulation Code. *Journal of Open Source Software* 5, 54 (2020), 2513. https://doi.org/10.21105/joss.02513
[6] Amazon Corp. 2019. Amazon EC2 G4 Instances with NVIDIA T4 Tensor Core GPUs, now available in 6 additional regions. https://aws.amazon.com/about-aws/whats-new/2019/10/amazon-ec2-g4-instances-with-nvidia-t4-tensor-core-gpus-now-available-in-6-additional-regions. [Accessed Aug 9, 2021].
[7] Ausar Geophysical. 2021. Wave propagation modules for PyTorch. https://github.com/ar4/deepwave. [Accessed Aug 9, 2021].

[8] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 163–174. https://doi.org/10.1109/ispass.2009.4919648

[9] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 71–81. https://doi.org/10.1109/P3HPC49587.2019.00012

[10] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, Vol. 4. Austin, TX, 1–7. https://doi.org/10.25080/majora-92bf1922-003

[11] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934* (2020).

[12] Jeffrey Burt. 2020. The Softer Side of Exascale. https://www.nextplatform.com/2020/02/14/the-softer-side-of-exascale. [Accessed Aug 9, 2021].

[13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[15] H Carter Edwards and Christian R Trott. 2013. Kokkos: Enabling performance portability across manycore architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*. IEEE, 18–24. https://doi.org/10.1109/xsw.2013.7

[16] Guin Gilman, Samuel S Ogden, Tian Guo, and Robert J Walls. 2021. Demystifying the Placement Policies of the NVIDIA GPU Thread Block Scheduler for Concurrent Kernels. *ACM SIGMETRICS Performance Evaluation Review* 48, 3 (2021), 81–88. https://doi.org/10.1145/3453953.3453972

[17] Google Corp. 2021. TensorBoard: TensorFlow's visualization toolkit. https://www.tensorflow.org/tensorboard. [Accessed Aug 9, 2021].

[18] Anton V Gorshkov, Michael Berezalsky, Julia Fedorova, Konstantin Levit-Gurevich, and Noam Itzhaki. 2019. GPU Instruction Hotspots Detection Based on Binary Instrumentation Approach. *IEEE Trans. Comput.* 68, 8 (2019), 1213–1224. https://doi.org/10.1109/tc.2019.2896628

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778. https://doi.org/10.1109/cvpr.2016.90

[20] Yangqing Jia. 2014. *Learning Semantic Image Representations at a Large Scale*. Ph.D. Dissertation. University of California, Berkeley, USA. http://www.eschol arship.org/uc/item/64c2v6sn

[21] Melanie Kambadur, Sunpyo Hong, Juan Cabral, Harish Patil, Chi-Keung Luk, Sohaib Sajid, and Martha A Kim. 2015. Fast computational gpu design with gt-pin. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 76–86. https://doi.org/10.1109/iiswc.2015.14

[22] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 473–486. https://doi.org/10.1109/isca45697.2020.00047

[23] Jeongnim Kim et al. 2018. QMCPACK: an open sourceab initioquantum Monte Carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter* 30, 19 (apr 2018), 195901. https://doi.org/10.1088/1361-648x/aab9c3

[24] Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten. 2013. Microarchitectural mechanisms to exploit value structure in SIMT architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 130–141. https://doi.org/10.1145/2508148.2485934

[25] Andreas Knüpfer, Christian Rössel, Dieter Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Competence in High Performance Computing 2011*. Springer Berlin Heidelberg, 79–91.

[26] William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. 2015. Improving CUDA DNA Analysis Software with Genetic Programming. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* (Madrid, Spain) *(GECCO '15)*. Association for Computing Machinery, New York, NY, USA, 1063–1070. https://doi.org/10.1145/2739480.2754652

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86. https://doi.org/10.1109/cgo.2004.1281665

[28] Alan Morris, Allen D. Malony, Sameer Shende, and Kevin Huck. 2010. Design and Implementation of a Hybrid Parallel Performance Measurement System. In *Proceedings of the 2010 39th International Conference on Parallel Processing (ICPP '10)*. IEEE Computer Society, Washington, DC, USA, 492–501.

[29] NVIDIA Corp. 2021. cuBLAS. https://developer.nvidia.com/cublas. [Accessed March 9, 2021].

[30] NVIDIA Corp. 2021. CUDA Graph: CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-graphs. [Accessed Aug 9, 2021].

[31] NVIDIA Corp. 2021. NVIDIA Compute Sanitizer API. https://docs.nvidia.com/cuda/sanitizer-docs/SanitizerApi/index.html. [Accessed Aug 9, 2021].

[32] NVIDIA Corp. 2021. NVIDIA CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit. [Accessed Aug 9, 2021].

[33] NVIDIA Corp. 2021. NVIDIA Jetson: The AI platform for autonomous everything. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems. [Accessed Aug 9, 2021].

[34] NVIDIA Corp. 2021. nvprof: CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/profiler-users-guide/index.html. [Accessed Aug 9, 2021].

[35] NVIDIA Corporation. 2021. *NVIDIA Nsight Compute*. https://developer.nvidia.com/nsight-compute [Accessed Aug 9, 2021].

[36] NVIDIA Corporation. 2021. *NVIDIA Nsight Systems*. https://developer.nvidia.com/nsight-systems [Accessed Aug 9, 2021].

[37] NVIDIA Corporation. 2021. *NVIDIA NVCC*. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html [Accessed Aug 9, 2021].

[38] OpenMP Architecture Review Board. 2021. *OpenMP Application Programming Interface*. https://www.openmp.org/spec-html/5.1/openmp.html [Accessed Aug 9, 2021].

[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).

[40] Yuanfeng Peng, Vinod Grover, and Joseph Devietti. 2018. Curd: A dynamic cuda race detector. *ACM SIGPLAN Notices* 53, 4 (2018), 390–403. https://doi.org/10.1145/3296979.3192368

[41] James C Phillips, David J Hardy, Julio DC Maia, John E Stone, João V Ribeiro, Rafael C Bernardi, Ronak Buch, Giacomo Fiorin, Jérôme Hénin, Wei Jiang, et al. 2020. Scalable molecular dynamics on CPU and GPU architectures with NAMD. *The Journal of chemical physics* 153, 4 (2020), 044130.

[42] Steve Plimpton. 1993. *Fast parallel algorithms for short-range molecular dynamics*. Technical Report. Sandia National Labs., Albuquerque, NM (United States).

[43] Jason Power, Joel Hestness, Marc S Orr, Mark D Hill, and David A Wood. 2014. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters* 14, 1 (2014), 34–36. https://doi.org/10.1109/lca.2014.2299539

[44] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/. [Accessed Aug 9, 2021].

[45] James Reinders. 2005. VTune Performance Analyzer Essentials. *Intel Press* (2005).

[46] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible software profiling of GPU architectures. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 185–197. https://doi.org/10.1145/2749469.2750375

[47] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. 2019. Redundant loads: A software inefficiency indicator. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 982–993. https://doi.org/10.1109/icse.2019.00103

[48] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 372–383. https://doi.org/10.1145/3352460.3358307

[49] Kai Wang and Calvin Lin. 2017. Decoupled Affine Computation for SIMT GPUs. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 295–306. https://doi.org/10.1145/3140659.3080205

[50] Benjamin Welton and Barton P. Miller. 2019. Diogenes: Looking for an Honest CPU/GPU Performance Measurement Tool. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 21, 20 pages. https://doi.org/10.1145/3295500.3356213

[51] Shasha Wen, Milind Chabbi, and Xu Liu. 2017. Redspy: Exploring value locality in software. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 47–61. https://doi.org/10.1145/3093336.3037729

[52] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. 2018. Watching for software inefficiencies with witch. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 332–347. https://doi.org/10.1145/3173162.3177159

[53] Shasha Wen, Xu Liu, and Milind Chabbi. 2015. Runtime value numbering: A profiling technique to pinpoint redundant computations. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 254–265. https:

//doi.org/10.1109/pact.2015.29

[54] Cort J Willmott and Kenji Matsuura. 2005. Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Climate research* 30, 1 (2005), 79–82. https://doi.org/10.3354/cr030079

[55] Ping Xiang, Yi Yang, Mike Mantor, Norm Rubin, Lisa R Hsu, and Huiyang Zhou. 2013. Exploiting uniform vector instructions for GPGPU performance, energy efficiency, and opportunistic reliability enhancement. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 433–442. https://doi.org/10.1145/2464996.2465022

[56] Chenle Yu, Sara Royuela, and Eduardo Quiñones. 2020. OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices. In *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*. 42–47. https://doi.org/10.1145/3378678.3391881

[57] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, et al. 2019. AMReX: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software* 4, 37 (2019), 1370–1370. https://doi.org/10.21105/joss.01370

[58] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. 2020. GVProf: A value profiler for GPU-based clusters. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16. https://doi.org/10.1109/SC41405.2020.00093

[59] Keren Zhou, Mark W Krentel, and John Mellor-Crummey. 2020. Tools for top-down performance analysis of GPU-accelerated applications. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12. https://doi.org/10.1145/3392717.3392752

[60] Keren Zhou, Xiaozhu Meng, Ryuichi Sai, and John Mellor-Crummey. 2021. GPA: A GPU Performance Advisor Based on Instruction Sampling. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 115–125. https://doi.org/10.1109/CGO51591.2021.9370339